Proceedings Paper:
Zhao, Shuai, Jie, Chen, Liang, Yaowei et al. (2026) Beyond Exact: Tight WCET Analysis of GPU Kernels with Branch Divergence. In: Design Automation Conference 2026: DAC 2026:Proceedings. Proceedings - Design Automation Conference. ACM.

https://doi.org/10.1145/3770743.3804186

Beyond Exact: Tight WCET Analysis of GPU Kernels with Branch Divergence

Shuai Zhao[1], Chen Jie[1], Yaowei Liang[1], Mingyue Cui[1], Xiaotian Dai[2], Nan Chen[2]
[1]Sun Yat-Sen University, China. [2]University of York, UK.

Abstract

The increasing adoption of GPUs in real-time systems necessitates precise timing analysis for GPU thread blocks to ensure overall system predictability. However, the uncertainties of branch executions within GPU warps impose significant barriers for predicting the worst-case execution time (WCET) of the thread block. Existing WCET analysis for thread blocks typically assumes deterministic warp execution paths, which is unrealistic given the dynamic control flows of threads within the warps. Moreover, as the warp scheduler operates as a black box, the analysis must rely on relaxed scheduling assumptions, resulting in overly-pessimistic bounds in order to cover edge scenarios. This paper first establishes the need for static analysis by showing how branch divergence can trigger timing anomalies and influence block WCETs. We then develop an exact WCET analysis for a GPU thread block under the same scheduling constraints as prior work while not imposing any warp execution path assumptions. Furthermore, by enforcing a practical constraint on warp executions, we present a tighter analysis that enhances system predictability. Experiments show that the proposed analysis under warp execution constraints significantly reduces the WCET estimations of GPU thread blocks (by 19.13% on average and up to 39.39%).

ACM Reference Format:
Shuai Zhao[1], Chen Jie[1], Yaowei Liang[1], Mingyue Cui[1], Xiaotian Dai[2], Nan Chen[2]. 2026. Beyond Exact: Tight WCET Analysis of GPU Kernels with Branch Divergence. In *63rd ACM/IEEE Design Automation Conference (DAC '26), July 26–29, 2026, Long Beach, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3770743.3804186

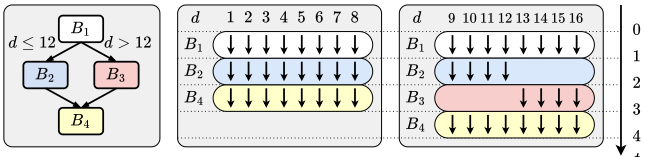
1 Introduction

The worst-case execution time (WCET) analysis is essential for ensuring the temporal correctness of real-time systems [9, 29]. For systems with CPUs, WCET analysis techniques are relatively mature [17, 21, 31, 32]. However, with the growing demand for artificial intelligence, graphics processing units (GPUs) are increasingly integrated into embedded and real-time systems [5, 6, 15]. Unlike CPUs, GPUs adopt the single-instruction multiple-thread (SIMT) execution model, where threads are grouped into warps that execute the same instruction in lockstep [8]. This fundamental difference renders CPU WCET analysis methods inapplicable to GPUs.

*Corresponding author: Nan Chen. Email: nan.chen@york.ac.uk.
This work is supported by National Key Research and Development Program under Grant 2024YFB4405600, National Natural Science Foundation of China under Grant 62302533, Guangdong Basic and Applied Basic Research Foundation under Grant 2024A1515010240, and FoShan NanHai key area research under Grant 2230032004606. Innovate UK SCHEME project under Grant 10065634.

DAC '26, Long Beach, CA, USA

ACM ISBN 979-8-4007-2254-7/2026/07
https://doi.org/10.1145/3770743.3804186



(a) CFG (b) without divergence (c) with divergence

Figure 1: Illustration of branch divergence under basic blocks in a CFG. In (a), d is a condition variable, where B_2 is executed if $d \leq 12$ and B_3 if $d > 12$. The execution time of every basic block is 1. In (b), all threads follow $d \leq 12$, resulting in the execution path $B_1 \rightarrow B_2 \rightarrow B_4$, with a warp execution time of 3. In (c), branch divergence occurs, with some threads executing B_2 and others executing B_3, leading to the path $B_1 \rightarrow B_2 \rightarrow B_3 \rightarrow B_4$ with a warp execution time of 4.

In the SIMT model, threads in a warp can take different paths, resulting in varying execution time of the warp. This occurs because when they encounter a conditional branch, the executions of the taken paths are serialized due to lockstep, causing their execution times to be accumulated and effectively extending the warp's overall execution path. This phenomenon is known as *branch divergence* [8], as illustrated in Fig. 1. Since the branch decisions for each thread cannot always be statically determined, branch divergence is conservatively handled by merging all possible branches into the warp execution path [14]. However, even if all warps are modeled by their worst-case execution paths, simulating these warps to obtain the overall WCET of a thread block remains infeasible, as their contention over shared resources (*e.g.*, pipelines and functional units) may prevent the worst-case scenario from occurring [24]. In practice, warps that take shorter paths may alter the order in which warps access shared resources, potentially introducing *timing anomalies* that prolong the completion time of the thread block [24] (see Sec. 3), thus emphasizing the necessity for static analysis that bounds the WCET among all possible warp execution paths.

Existing GPU WCET analysis for a thread block [13] avoids the branch divergence by assuming a specific execution path for each warp. However, this assumption is impractical, as it requires precise knowledge of per-thread branch conditions to determine warp-level control flow. In addition, due to the proprietary nature of GPU architectures, the warp scheduling mechanism remains opaque, introducing uncertainty in warp execution. To ensure correctness, existing analysis [13] relies on overly-relaxed scheduling assumptions (*e.g.*, work-conserving only) and considers the worst case over all possible warp execution orders. This can lead to pessimistic or even impractical results. Thus, addressing the uncertainties from branch divergence while delivering tight bounds is particularly challenging for the WCET analysis of GPU thread blocks.

Contribution. This paper first addresses the importance of static analysis by providing an in-depth analysis of timing anomalies caused by branch divergence, identifying the conditions under

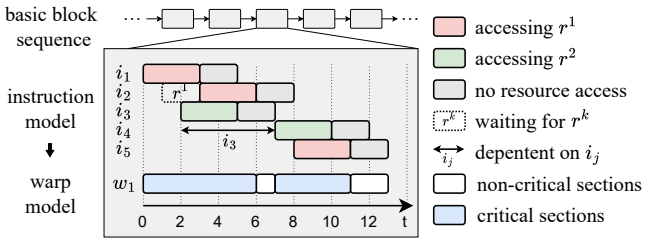


Figure 2: Construction of instruction and warp model.

which they occur and their impact on WCETs. Building on this, we propose an exact WCET analysis of GPU thread block under the same scheduling constraint as the existing work [13], while not imposing any assumptions of the warp execution paths. More importantly, we demonstrate that by enforcing a practically implementable constraint on the warp execution order, a much tighter WCET analysis can be constructed, which can improve the system predictability. Experimental results show that compared to the exact analysis for any work-conserving schedule, the analysis with additional warp execution constraints reduces the WCET estimations by 19.13% on average (up to 39.39%) for both synthetic workloads and the Rodinia benchmark suite [2].

2 Preliminaries

As with [13], we analyze a GPU kernel executed by a thread block consisting of M warps, which are scheduled under a work-conserving scheme on a single sub-core. Since warps do not migrate across sub-cores during kernel execution, the proposed analysis can be effectively extended to multiple sub-cores [13]. Finer-grained timing analysis for multiple blocks is postponed to future work.

A sub-core in a stream multiprocessor consists of multiple functional units (FUs) for executing the instruction-level operations, *e.g.*, the Floating-Point Units and Special Function Units. When an instruction accesses an FU, it exclusively occupies it for execution, during which other instructions attempting to access the same FU are blocked. We refer to such an FU as a resource, denoted by r^k. Notation $\mathcal{R}$ denotes the set of resources in a sub-core.

A GPU kernel function can be represented as a Control Flow Graph (CFG) that is constructed by the compiler, *e.g.*, LLVM [18]. For loop structures, we assume that the loop bounds are specified by static analysis [25, 27, 28] or the engineers. As for branches, when threads within a warp take different control paths, the warp is assumed to execute these branches serially, as shown in Fig. 1(c). Thus, each warp follows a control path consisting of a sequence of basic blocks, as illustrated at the top of Fig. 2.

Following the approach in [13], Sec. 2.1 first models the behavior of the instructions in a basic block. Based on the instruction model, we abstract the execution sequence of a warp executing the instructions in one basic block, and further model the kernel execution for a sequence of basic blocks (Sec. 2.2). These models serve as the foundation for the proposed analysis. Finally, we introduce the motivation of this paper (Sec. 2.3).

2.1 Instruction Model

As with [13], for the instructions in a basic block (executed by threads in a warp in lockstep), an instruction model can be constructed according to the GPU pipeline (see Fig. 2). Each instruction i_j in a basic block performs an operation defined by its operator and operands. The operator defines the computation, while the operands

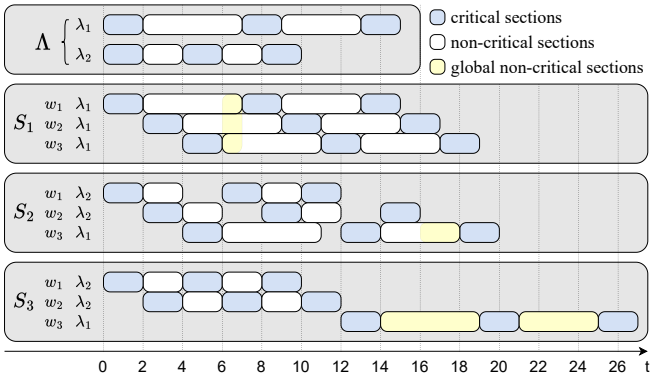


Figure 3: Three possible execution scenarios in which each of the three warps follows a path $\lambda_m \in \Lambda$ under a work-conserving scheduling scheme.

represent the registers or memory locations accessed. For example, in the instruction `sin.approx.f32 reg_out, reg_in`, the operator computes an approximate sine of the value in `reg_in` using Special Function Unit (SFU), and stores the result into `reg_out`.

For two instructions i_a and i_b (with $a \neq b$), i_a depends i_b if any operand of i_a requires the result produced by i_b. Such dependencies define the partial order of instruction issues within the same warp, which can be obtained via static analysis. During execution, additional timing parameters are introduced to describe the behavior of instructions. The time point at which i_j is sent by the warp scheduler is referred to as the *issue time*, denoted as $i_j.issue$, while the time point at which i_j begins execution on its required resource is referred to as the *start time*, denoted as $i_j.start$. Following prior studies [1, 3, 13], we assume that at most one instruction can be issued per cycle, and memory-related contention is excluded from consideration. As with [13], the execution characteristics of instructions are described as follows.

I-C1. The instructions are issued in program order and start once the required resources become available, which then executes to completion without further stalls.

I-C2. When an instruction occupies a resource, no other instruction requiring the same resource can start.

I-C3. An instruction i_j may be issued only after the completion of all instructions on which it depends.

Based on the characteristics, $i_j.issue$ and $i_j.start$ are governed by the following rules. (i) Since instructions are issued in program order, $i_j.issue < i_{j+1}.issue$ for all instructions (by I-C1). (ii) For an instruction i_j requiring resource r^k, i_j will access r^k for a duration starting from $i_j.start$, during which no other instruction requiring r^k can start (by I-C2). (iii) i_j will continue executing for a duration after $i_j.start$ and then complete. Moreover, i_j cannot be issued until all instructions on which i_j depends have completed (by I-C3). These rules determine the execution of instructions in a warp.

Example 1. The instruction model in Fig. 2 illustrates the execution of five instructions in a basic block, where i_1, i_2, and i_5 require r^1, while i_3 and i_4 require r^2, and i_4 depends on i_3. For all instructions, we assume that the resource access time and execution time are 3 and 5, respectively. At time 0, i_1 starts with $i_1.issue = i_1.start = 0$. Then, i_1 occupies r^1 until time 3 and completes at time 5. For i_2, we have $i_2.issue = 1$ (*i.e.*, single-issue), but it

must wait until r^1 is released so that $i_2.start = 3$; it then occupies r^1 until time 6 and completes at time 8. Similarly, we have $i_3.issue = i_3.start = 2$, which occupies r^2 until time 5 and completes at time 7. For i_4, since i_4 depends on i_3, we have i_4.issue = i_4.start = 7 (i.e., after i_3 completes). It occupies r^2 until time 10 and completes at time 12. Finally, $i_5.issue = i_5.start = 8$, where i_5 occupies r^1 until time 11 and completes at time 13.

2.2 Warp Model

Based on the instruction model, the execution of a warp can be abstracted as a sequence of alternating *critical* and *non-critical* sections following the method in [13]. An interval is a *critical* section if any resource in $\mathcal{R}$ is exclusively occupied by a thread in the warp, whereas a *non-critical* section is an interval at which no resource in $\mathcal{R}$ is occupied by any thread. This abstraction yields the warp model illustrated in Fig. 2. Although this abstraction may overestimate execution time when multiple warps execute concurrently, it does not affect the analysis discussed in Sec. 3 and 4.

For a warp (denoted as w_u), let $z^*_{u,h}$ and $z_{u,h}$ denote its h-th critical and non-critical sections, respectively. For the instructions in Fig. 2, two critical sections (*i.e.*, $z^*_{1,1} = [0, 6)$, $z^*_{1,2} = [7, 11)$) and two non-critical sections (*i.e.*, $z_{1,1} = [6, 7)$, $z_{1,2} = [11, 13)$) are constructed, as shown in the warp model of the figure.

In addition, as discussed in Sec. 1, a warp may follow different execution paths due to branch divergence. To describe these paths formally, let λ_m denote a path represented by a sequence of alternating sections $z^*_{u,h}$ and $z_{u,h}$, and let Λ denote the set of all possible paths that a warp may take. Notations $Z^*_{\lambda_m}$ and Z_{λ_m} denote the critical and non-critical sections in a path λ_m, respectively (*e.g.*, $\Lambda = \{\lambda_1, \lambda_2\}$, $Z^*_{\lambda_1} = \{[0, 2), [7, 9), [13, 15)\}$ and $Z^*_{\lambda_2} = \{[0, 2), [4, 6), [8, 10)\}$ in Fig. 3).

Next, we describe the concurrent execution scenarios of warps. A *scenario* S_x represents a concrete execution trace of M warps, where each warp follows a specific path $\lambda_m \in \Lambda$. In a scenario S_x, its WCET R_x is defined as the response time of the warp that finishes last. For example, Fig. 3 illustrates three execution scenarios for the case $M = 3$. In S_1, the response times of w_1, w_2, and w_3 are 15, 17, and 19, respectively; thus, we have $R_1 = 19$.

2.3 Motivation

The existing analysis [13] derives an upper bound on the WCET of a thread block by calculating the worst-case finish time for each warp and taking the maximum value. For a warp w_u, its worst-case finish time is computed as the sum of its critical and non-critical sections, plus the critical sections of other warps within the block as interference. However, this bound is derived based on an impractical assumption that the execution path of each warp must be known a priori, while suffering from significant pessimism due to overly-relaxed assumptions (see Sec. 1). To address this, Sec. 3 presents an exact WCET analysis for thread blocks by considering all possible warp execution scenarios under branch divergence. Then, Sec. 4 demonstrates that a tighter upper bound can be achieved by enforcing a practically realizable constraint on warp execution.

3 Constructing the Exact Bound

This section first describes the timing anomalies induced by branch divergence, which necessitates the static analysis. It then presents an exact WCET analysis for GPU thread blocks under the same scheduling constraints as prior work, without assumptions of specific warp execution paths.

3.1 WCET Bound without Deterministic Paths

For a scenario S_x, branch divergence may occur during the execution of the warps, which can cause some warps to follow a path that is shorter than the longest path $\lambda^\dagger$. However, this may cause a *timing anomaly* [24] if its overall WCET exceeds that of the scenario where all warps follow the $\lambda^\dagger$. We define a scenario as $S^\dagger$ if all w_u follow the longest path $\lambda^\dagger$ and denote the WCET of $S^\dagger$ as $R^\dagger$. In S_x, if R_x exceeds $R^\dagger$, then a timing anomaly is said to occur in S_x. As illustrated in Fig. 3, $\lambda^\dagger = \lambda_1$, $S^\dagger = S_1$ and $R^\dagger = 19$. However, S_2 and S_3 yield $R_2 = 20 > R^\dagger$ and $R_3 = 27 > R^\dagger$, respectively. Hence, timing anomalies occur in both S_2 and S_3.

To cover the impact of timing anomalies, providing a static analysis of the overall WCET across all scenarios is necessary. For convenience, we define the *global non-critical section* (denoted as $z^\neg_h$) as the interval in S_x during which all unfinished warps are concurrently executing their non-critical sections. The sets of all $z^*_{u,h}$ and $z^\neg_h$ in S_x are denoted by $Z^*_{S_x}$ and $Z^\neg_{S_x}$, respectively. By definition, $Z^\neg_{S_x}$ and $Z^*_{S_x}$ are mutually exclusive within S_x. For example, As illustrated in Fig. 3, $Z^*_{S_1} = \{[0, 6), [7, 19)\}$ and $Z^\neg_{S_1} = \{[6, 7)\}$, which yields $Z^\neg_{S_1} \cap Z^*_{S_1} = \emptyset$. Let function $C(\cdot)$ denote the total duration of the given intervals. We can obtain the WCET under any scenario through the total durations of its critical sections and global non-critical sections, as shown in Lemma 1.

Lemma 1. *For any scenario S_x, the R_x equals the sum of the total duration of its critical sections and global non-critical sections, i.e., $R_x = C(Z^*_{S_x}) + C(Z^\neg_{S_x})$.*

Proof. For a S_x, since $Z^\neg_{S_x}$ and $Z^*_{S_x}$ are mutually exclusive, and for any time point $t \in [0, R_x)$, either $t \in Z^*_{S_x}$ or $t \in Z^\neg_{S_x}$, it follows that $R_x = C(Z^*_{S_x}) + C(Z^\neg_{S_x})$. □

As an example of Lemma 1, consider Fig. 3, where $C(Z^*_{S_1}) = 18$, $C(Z^\neg_{S_1}) = 1$, and $R_1 = C(Z^*_{S_1}) + C(Z^\neg_{S_1}) = 19$. Since $\lambda^\dagger$ represents the longest path, it follows that $C(Z^*_{\lambda^\dagger}) \geq C(Z^*_{\lambda_m})$ for any $\lambda_m \in \Lambda$, which leads to $C(Z^*_{S^\dagger}) \geq C(Z^*_{S_x})$ for any scenario S_x. Thus, based on Lemma 1, if a timing anomaly occurs in S_x (*i.e.*, $R_x > R^\dagger$), it must be due to $C(Z^\neg_{S_x}) > C(Z^\neg_{S^\dagger})$. For example, in Fig. 3, $C(Z^\neg_{S^\dagger}) = 1$, $C(Z^\neg_{S_2}) = 2$, and $C(Z^\neg_{S_3}) = 9$, which lead to $R_2 > R^\dagger$ and $R_3 > R^\dagger$.

To handle such anomalies, an upper bound on $C(Z^\neg_{S_x})$ must be determined for each scenario S_x. Let $Z^*_{w^\diamond}$ and $Z_{w^\diamond}$ denote the sets of all critical and non-critical sections of the last warp completed within S_x, respectively. Lemma 2 shows that the total non-critical duration of the last completed warp in S_x provides a bound on the duration of the global non-critical sections of S_x. The proof of exactness is presented in Sec. 3.2.

Lemma 2. *For any scenario S_x, the duration of its global non-critical sections is bounded by the total non-critical duration of the last completed warp, i.e., $C(Z^\neg_{S_x}) \leq C(Z_{w^\diamond})$.*

Proof. If $C(Z^\neg_{S_x}) > C(Z_{w^\diamond})$, then there exists an interval $z \subseteq Z^\neg_{S_x}$ such that $z \subseteq [0, R_x)$, $z \notin Z_{w^\diamond}$, and $z \notin Z^*_{w^\diamond}$. Therefore, for any time point $t \in z$, either $w^\diamond$ is blocked by another warp executing its critical section, or $w^\diamond$ has already completed. Since $Z^\neg_{S_x}$ and $Z^*_{S_x}$ are

mutually exclusive, we have $z \notin Z^*_{S_x}$. Thus, at time point t, no warp is executing in the critical section, implying that $w^\diamond$ has already completed. Furthermore, since the response time of $w^\diamond$ equals R_x, this contradicts the fact that $z \subseteq [0, R_x)$. □

As an example of Lemma 2, consider Fig. 3. For S_1, $C(Z^\neg_{S_1}) = 1 < C(Z_{w^\diamond}) = 9$; for S_2, $C(Z^\neg_{S_2}) = 2 < C(Z_{w^\diamond}) = 9$; and the same holds for S_3. However, the exact scheduling order of warps is generally unpredictable, making it difficult to determine which warp will complete the last. Since each warp in any scenario S_x follows a path selected from Λ, $\max_{\lambda_m \in \Lambda} C(Z_{\lambda_m})$ provides an upper bound for $C(Z^\neg_{S_x})$ in all scenarios. Similarly, for a scenario in which M warps execute concurrently, the upper bound for $C(Z^*_{S_x})$ is derived as $\max_{\lambda_m \in \Lambda} C(Z^*_{\lambda_m}) \times M$. Thus, the WCET $\overline{R}$ for any scenario under work-conserving scheduling can be formally expressed as Eq. 1.

$$\overline{R} = \max_{\lambda_m \in \Lambda} C(Z^*_{\lambda_m}) \times M + \max_{\lambda_m \in \Lambda} C(Z_{\lambda_m}) \tag{1}$$

In practice, the number of paths in the set Λ is often extremely large, making exhaustive traversal infeasible. However, since the longest path $\lambda^\dagger$ corresponds to serial execution of all branches, we have $C(Z^*_{\lambda^\dagger}) \geq C(Z^*_{\lambda_m})$ and $C(Z_{\lambda^\dagger}) \geq C(Z_{\lambda_m})$ for any $\lambda_m \in \Lambda$. Consequently, Eq. 1 is transformed into Eq. 2. For example, in Fig. 3, $M = 3$, $\lambda^\dagger = \lambda_1$, $C(Z^*_{\lambda^\dagger}) = 6$ and $C(Z_{\lambda^\dagger}) = 9$, which lead to $\overline{R} = 6 \times 3 + 9 = 27$.

$$\overline{R} = C(Z^*_{\lambda^\dagger}) \times M + C(Z_{\lambda^\dagger}) \tag{2}$$

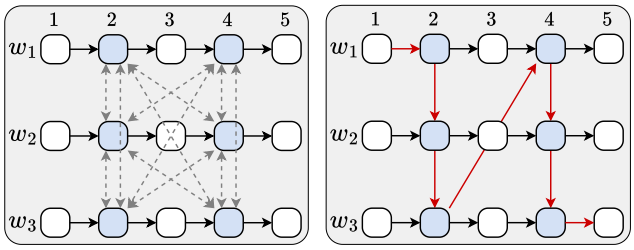
The above analysis neither relies on a fixed static path for each warp nor requires enumerating all paths in Λ, thereby deriving an efficient bound across diverse scenarios, as expressed in Eq. 2. Unlike existing analyses [13], our approach explicitly considers all possible execution paths that may arise from divergence and identifies the scenario that produces the worst-case timing behavior of the thread block. Once this scenario is determined, the remaining timing evaluation can proceed in a manner similar to [13], but now grounded in the scenario that indeed produces the worst case.

3.2 Exactness of the Bound

We now prove the exactness of the constructed analysis under any work-conversing schedule, *i.e.*, the only scheduling constraint adopted in [13]. As described in [12, 16], a timing bound is *exact* if there always exists an execution scenario such that its WCET equals the derived bound. Theorem 1 describes the exactness of $\overline{R}$ under any work-conserving schedule.

Theorem 1. *Under work-conserving scheduling constraints, there always exists a scenario S_x such that $R_x = \overline{R}$.*

Proof. Consider a scenario S_x with M warps. When $M = 1$, $R_x = \overline{R}$ holds trivially. For $M \geq 2$, since each warp may follow a path shorter than $\lambda^\dagger$ due to branch divergence, we construct the execution paths of M warps satisfying the following conditions: (i) one warp executes $\lambda^\dagger$; (ii) each of the remaining $M-1$ warps executes a path λ_m such that $C(Z^*_{\lambda_m}) = C(Z^*_{\lambda^\dagger})$; and (iii) these $M-1$ warps can form a scenario S_0 where $C(Z^\neg_{S_0}) = 0$. Since the schedule is constrained only by work-conserving behavior, we can construct a scenario S_x in which the $M-1$ warps execute according to S_0, causing the warp executing $\lambda^\dagger$ to remain blocked without violating work-conserving constraints. The duration of this phase is $C(Z^*_{\lambda^\dagger}) \times (M-1)$. After the other warps complete, the warp executing $\lambda^\dagger$ executes with a duration of $C(Z^*_{\lambda^\dagger}) + C(Z_{\lambda^\dagger})$. Hence, $R_x = C(Z^*_{\lambda^\dagger}) \times M + C(Z_{\lambda^\dagger}) = \overline{R}$. □



(a) $G(V, E)$ represents the constraints under any work-conserving scheduling. (b) $G(V, E_{\text{dag}})$ represents constraints with a topological order of warps.

Figure 4: Two scheduling constraints represented by graph models. ***(The blue and white nodes represent critical and non-critical sections, with execution times of 1 and 2, respectively. The edges denote the dependencies of nodes. The red path in (b) indicates the longest path of the DAG.)***

For example, the scenario S_3 in Fig. 3 illustrates the case where $R_x = \overline{R}$ under work-conserving scheduling constraints. As shown in the figure, w_3 executes $\lambda^\dagger$, $C(Z^*_{\lambda_2}) = C(Z^*_{\lambda^\dagger})$, and the interval $[0, 12)$ contains no global non-critical section. Consequently, w_3 can start execution only after w_1 and w_2 complete, resulting in $R_3 = 27 = \overline{R}$.

Under a work-conserving scheduler, critical sections may execute in many different orders as long as mutual exclusion is maintained. Some of these valid orders are significantly worse than others, requiring the analysis to account for the worst-case scenario. As a result, although being exact in theory, the bound presented in Eq. 2 remains conservative in practice. However, by introducing constraints that restrict the admissible execution orders of the warps, the number of possible execution scenarios (along with the corner cases) can be effectively reduced, thereby leading to a tighter and more realistic bound (see Sec. 4).

4 Constructing the Tighter Bound

As discussed in Sec. 3.2, the bound presented in Eq. 2 is conservative due to the overly-relaxed scheduling assumption (*i.e.*, work-conserving only). To further tightening the bound, we model the scheduling constraints that restrict the admissible execution orders of critical sections, hencing making execution of warps more predictable (Sec. 4.1). With the execution constraints introduced, we show that a tighter analysis can be constructed following the enforced execution order of the warps (Sec. 4.2).

4.1 Graph Model with Scheduling Constraints

We first construct a graph model to describe the execution characteristics of the work-conserving scheduling constraint more concretely. In this model, the nodes represent the critical and non-critical sections of the M warps executing $\lambda^\dagger$. Specifically, directed edges between nodes indicate that the source node must be executed before the target node, while bidirectional edges indicate that either node can be executed first. Thus, the edges represent the execution order between warps. This graph model is denoted as $G(V, E)$, where V and E represent the sets of nodes and edges, respectively.

The scheduling constraints in the existing analysis [13] are represented as the graph $G(V, E)$, which allows warps to follow any scheduling policy under a work-conserving scheduler. Hence, $G(V, E)$ can be effectively constructed by adding the following edges to the critical and non-critical nodes of the warps in two steps: (i) a directed edge between every two adjacent critical section $z^*_{u,h}$ and non-critical $z_{u,h'}$ of each warp w_u, representing the predefined execution order within a warp; (ii) a bidirectional edge between every two critical sections $z^*_{u,h}$ and $z^*_{u',h'}$ of different warps (*i.e.*, $u \neq u'$), indicating their mutual exclusion during execution.

Example 2. Fig. 4(a) illustrates the graph $G(V, E)$ built from three warps, each consisting of five nodes arranged in a 3×5 matrix. Each warp executes two critical sections and three non-critical sections, with directed edges connecting adjacent nodes within the same warp to preserve the intra-warp execution order. For the critical sections of those warps, bidirectional edges are added to connect every pair of critical sections across different warps, indicating that no fixed execution order is imposed between them.

In $G(V, E)$, the large number of bidirectional edges permits many possible execution orders among warps, leading to a pessimistic upper bound derived from $G(V, E)$. Therefore, to restrict the admissible execution orders, we enforce a specific topological order for the critical nodes across different warps. This effectively transforms $G(V, E)$ into a directed acyclic graph (DAG) by converting the bidirectional edges into directed edges while removing redundant ones. Before presenting the approach for realizing such a constraint, we first provide an example that illustrates the transformation process and highlights its advantages.

Example 3. Fig. 4(b) illustrates the same warps as in Fig. 4(a) that follows a specific topological order. In particular, the bidirectional edges are replaced with directed ones as follows: (i) For every two adjacent critical sections in each column, a directed edge is added from the critical section with a smaller warp index to the one with the larger index. (ii) For critical sections in different columns, a directed edge is added from the last critical section in the earlier column to the first one in the later column. By doing so, a DAG is constructed from the warps in Fig. 4(a) that enforces a specific execution order for the critical sections. Compared to the graph in Fig. 4(a), the bound on the worst-case makespan of the DAG (*i.e.*, the block WCET) is reduced by 16.7% (from 12 in Fig. 4(a) to 10 in Fig. 4(b) as shown by the example).

We now present the rules that transfer a given graph $G(V, E)$ into a DAG, represented as $G(V, E_{\text{dag}})$. Specifically, for every bidirectional edge between two critical sections in $G(V, E)$ (say $z^*_{u,h}$ and $z^*_{u',h'}$), it is replaced according to the following rules:

Rule 1. If $h = h'$ with $u < u'$, the bidirectional edge becomes a directed edge connected from $z^*_{u,h}$ to $z^*_{u',h'}$.

Rule 2. If $h < h'$, the bidirectional edge becomes a directed edge connected from $z^*_{u,h}$ to $z^*_{u',h'}$ with $u > u'$.

With the Rules 1 and 2 applied for every two critical sections in $G(V, E)$, the DAG $G(V, E_{\text{dag}})$ is constructed, enforcing a topological sorting of all the sections. During execution, the topological order specifies the order of the critical sections accessing the resources. This significantly limits possible execution scenarios (especially corner cases), thereby leading to a tighter bound.

In real-world GPU applications, schedulers may not always support policies that enforce such topological constraints. In such cases, the constraints on critical sections can be realized through code modifications or compiler instrumentation [20]. Specifically, semaphores can be applied to specify the order of critical sections in the same column (*i.e.*, Rule 1) [1], while synchronization can be employed to enforce the execution order across the columns (*i.e.*, Rule 2) [4]. Similar approaches have been explored in [11, 19, 23] for GPU timing analysis or investigating new scheduling strategies. Although these mechanisms may introduce additional overhead, they preserve a key characteristic for the systems, *i.e.*, enhanced predictability with tighter bounds. In addition, we note that further optimizations are possible and will be investigated in future work.

4.2 WCET Bound based on DAG

We now present the analysis for the worst-case makespan for the constructed DAG $G(V, E_{\text{dag}})$, which has been investigated in many prior works [7, 10, 33]. However, in our context, the maximum parallelism of the DAG at any time point is effectively bounded by the number of warps M, *i.e.*, when they are all executing a non-critical section. In addition, since the M warps can be executed in parallel in a sub-core, the finish time of every node is determined solely by the finish time of its predecessors and the execution time of itself, without incurring any interference from the parallel nodes [10, 33]. Under this case, the makespan R for the DAG under any scenario is effectively bounded by the longest path of the DAG, *i.e.*, the path with the maximum accumulated execution time [33].

Following the above, let $\mathcal{V}$ denote the set of all critical and non-critical sections in the longest path of $G(V, E_{\text{dag}})$, Eq. 3 presents the bound of the WCET for a thread block with the topological order imposed on its warps.

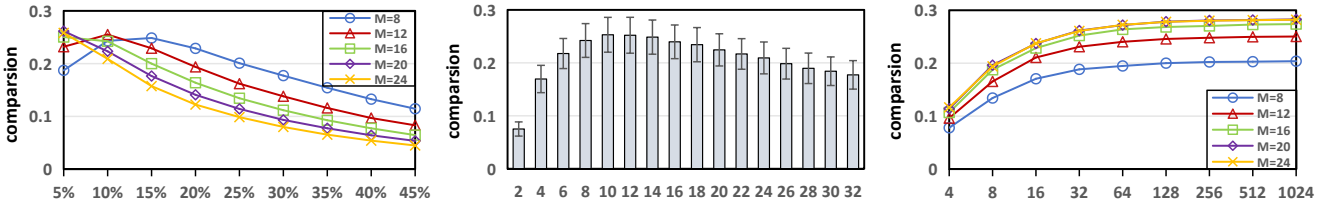
$$\widetilde{R} = \sum_{v \in \mathcal{V}} C(v) \tag{3}$$

With Eq. 3, a tighter analysis is constructed compared to the one introduced in the previous section (see Sec. 5 for results). We note that as discussed in [33], this analysis is sustainable, *i.e.*, nodes executing less than their execution times would not result in a makespan higher than $\widetilde{R}$. This justifies that under any execution scenario S_x of a block with the topological order enforced, the WCET is effectively bounded by $\widetilde{R}$. In addition, for any scheduling policy that follows a topological order among warp executions, a tight upper bound can be derived using a similar approach.

Finally, it is worth noting that our primary focus is to improve system predictability through static analysis. The execution constraints introduced in Sec. 4.1 are thereby examined analytically for deriving a tighter WCET bound. The implementation and the resulting performance are beyond the scope of this paper. Instead, our contribution lies in highlighting the theoretical benefits of the introduced constraints within timing analysis, leading to a predictable GPU system that is suitable for safety-critical domains.

5 Evaluation

This section evaluates the timing bounds $\overline{R}$ and $\widetilde{R}$ developed in Sec. 3 and Sec. 4, respectively. For a kernel, we evaluate $\widetilde{R}$ and $\overline{R}$ by the metric $(\overline{R} - \widetilde{R})/\overline{R}$ (*i.e.*, the tightness of $\widetilde{R}$ compared to $\overline{R}$). We first conduct evaluations of the analysis using synthetic thread blocks (Sec. 5.1) under the following parameters: the number of warps (M),



(a) N = 32 with varied c-ratio (%). (b) c-ratio = 10%, N = 32 with varied M. (c) c-ratio = 5% with varied N.

Figure 5: Comparison between $\overline{R}$ and $\widetilde{R}$ with varied M, N, and c-ratio.

the number of nodes in $\lambda^{\dagger}$ (denoted as N), and the ratio of the total duration of all critical sections to the overall execution time of $\lambda^{\dagger}$ (referred to as the *c-ratio*). Then, we obtain the hardware parameters of NVIDIA Quadro K620 following the method in [28], and conduct comparisons on 22 kernels from the Rodinia benchmark suite [2], demonstrating the practical effectiveness of our approach (Sec. 5.2).

5.1 Comparisons under Synthesized Workload

Experimental setup. We developed a lightweight simulator to generate synthetic thread blocks for evaluation. A thread block is constructed by generating a path $\lambda^{\dagger}$, as described below. The first node in $\lambda^{\dagger}$ is randomly assigned as either critical or non-critical. Then, the number of critical and non-critical sections is determined accordingly based on the total number of nodes N. The overall execution time of $\lambda^{\dagger}$ is fixed at 1000. The average execution time of critical and non-critical sections is calculated based on both a given c-ratio and the number of critical and non-critical sections. The duration of each critical or non-critical section is randomly selected between 0 and the corresponding average duration while maintaining the c-ratio. With $\lambda^{\dagger}$, M warps are generated for constructing the thread block. For each configuration, 1000 thread blocks are generated and analyzed, with the average of $(\overline{R} - \widetilde{R})/\overline{R}$ reported.

Results. As illustrated in Fig. 5(a), for fixed M and N, the difference between $\widetilde{R}$ and $\overline{R}$ in general decreases as the c-ratio increases. This is because with a lower c-ratio (*i.e.*, with longer non-critical sections), the bound $\widetilde{R}$ achieves tighter results by effectively capturing the parallelism between the non-critical sections. This observation is more obvious with a higher number of warps M, where the bound $\widetilde{R}$ provides the tightest results when the c-ratio is close to 0. In Fig. 5(b), the difference between $\widetilde{R}$ and $\overline{R}$ initially increases rapidly with the increase of M, but then gradually decreases. This trend occurs because when c-ratio = 10%, a large M also increases the portion of critical sections that undermines the parallelism of the warps, thereby reducing the difference between $\widetilde{R}$ and $\overline{R}$. As shown in Fig. 5(c) with c-ratio = 5%, the difference between $\widetilde{R}$ and $\overline{R}$ grows steadily with the increase of N. This observation confirms that the bound $\widetilde{R}$ is significantly tighter than $\overline{R}$ when the c-ratio is relatively small, even if the number of critical sections increases in each warp.

5.2 Comparisons under Benchmarks

Experimental setup. As with [28], to evaluate the Rodinia benchmark suite, we use a Quadro K620 GPU with the Maxwell GM107 architecture, operating at a clock frequency of 1072 MHz. The arithmetic and memory latencies (*i.e.*, execution time of a critical and non-critical section) are measured using micro-benchmarks [22, 26, 30]. Following the method in [28], we calculate the number of warps of a block based on both hardware constraints and the program configurations, where each kernel is executed by a single block on a sub-core. We process the kernel source code following Sec. 2, and perform both analysis to obtain both $\overline{R}$ and $\widetilde{R}$, respectively.

Table 1: Comparison of $\overline{R}$ and $\widetilde{R}$ for different kernels.

Kernel	c-ratio (%)	$\overline{R}$ (s)	$\widetilde{R}$ (s)	comparison (%)
findK	3.78	11.50	8.68	24.52
findRangeK	3.33	21.00	14.99	28.62
bpnn_adjust_weights_ocl	3.41	0.33	0.24	27.27
bpnn_layerforward_ocl	3.09	0.16	0.14	12.50
BFS_1	4.63	7.80	6.39	18.08
BFS_2	5.19	0.82	0.72	12.20
compute_flux	13.73	2.74	2.44	10.95
compute_step_factor	14.20	0.18	0.17	5.56
initialize_variables	8.58	0.16	0.13	18.75
time_step	25.70	0.35	0.33	5.71
Fan1	6.62	4.45×10^{-3}	3.77×10^{-3}	15.28
Fan2	8.63	17.24	14.07	18.39
kmeans_kernel_c	8.26	1.63	1.59	2.45
kmeans_swap	7.01	0.32	0.22	31.25
lud_diagonal	9.01	331.10	263.60	20.39
lud_internal	7.83	14.68	11.72	20.16
lud_perimeter	8.09	446.50	342.90	23.20
NearestNeighbor	4.24	0.61	0.44	27.87
nw_kernel1	6.69	0.94	0.74	21.28
findIndexSeq	6.82	21.20	15.05	29.01
particle_kernel	4.59	129.20	78.31	39.39
dynproc_kernel	3.68	5.52	5.08	7.97
average	7.60	–	–	19.13

Results. As shown in Tab. 1, the c-ratio for most kernels ranges from 3% to 15%, with an average of 7.6%, indicating that the ratio of critical sections is relatively small for most real-world kernels. This further justifies the effectiveness of the bound $\widetilde{R}$, as it provides substantially tighter results with a small c-ratio. For each kernel, the bound $\widetilde{R}$ derived in Sec. 4 is consistently tighter than $\overline{R}$ derived in Sec. 3, *i.e.*, by 19.13% on average and up to 39.39%. This again demonstrates the effectiveness of the analysis proposed in Sec. 4.

6 Conclusion

This paper presents a comprehensive study of the WCET analysis for GPU thread blocks. We first describe the need for an analysis that covers various warp execution scenarios by highlighting potential timing anomalies due to branch divergence. Then, an exact WCET analysis is proposed following the same scheduling assumptions of the existing analysis, while not imposing any warp execution path assumptions. Furthermore, we show that by enforcing a practical warp scheduling constraint, a tighter bound can be established that enhances system predictability. Experiments show that the analysis with additional scheduling constraints significantly tightens the WCET of thread blocks.

References

[1] Tor M Aamodt, Wilson Wai Lun Fung, Timothy G Rogers, and Margaret Martonosi. 2018. *General-purpose graphics processor architectures*. Springer.

[2] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.

[3] Noïc Crouzet, Thomas Carle, and Christine Rochange. 2024. Coordinating the fetch and issue warp schedulers to increase the timing predictability of GPUs. In *2024 27th Euromicro Conference on Digital System Design (DSD)*. IEEE, 343–350.

[4] Preyesh Dalmia, Rohan Mahapatra, Jeremy Intan, Dan Negrut, and Matthew D Sinclair. 2022. Improving the scalability of GPU synchronization primitives. *IEEE Transactions on Parallel and Distributed Systems* 34, 1 (2022), 275–290.

[5] Glenn A Elliott and James H Anderson. 2012. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems* 48, 1 (2012), 34–74.

[6] Glenn A Elliott, Bryan C Ward, and James H Anderson. 2013. GPUSync: A framework for real-time GPU management. In *2013 IEEE 34th Real-Time Systems Symposium*. IEEE, 33–44.

[7] Ronald L. Graham. 1969. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics* 17, 2 (1969), 416–429.

[8] Axel Habermaier and Alexander Knapp. 2012. On the correctness of the SIMT execution model of GPUs. In *European Symposium on Programming*. Springer, 316–335.

[9] Sebastian Hahn. 2018. On static execution-time analysis. (2018).

[10] Qingqiang He, Nan Guan, Mingsong Lv, Xu Jiang, and Wanli Chang. 2022. Bounding the response time of DAG tasks using long paths. In *2022 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 474–486.

[11] Yijie Huangfu and Wei Zhang. 2018. WCET Analysis of GPU L1 Data Caches. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2018.8547582

[12] Joxan Jaffar, Andrew E Santosa, and Razvan Voicu. [n. d.]. Exact WCET Analysis over Symbolic Explicit Paths. ([n. d.]).

[13] Louison Jeanmougin, Thomas Carle, and Christine Rochange. 2025. Bounding the WCET of a GPU Thread Block with a Multi-Phase Representation of Warps Execution. In *37th Euromicro Conference on Real-Time Systems (ECRTS 2025)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 11–1.

[14] Louison Jeanmougin, Thomas Carle, Pascal Sotin, and Christine Rochange. 2023. Warp-level CFG construction for GPU kernel WCET analysis. In *21st International Workshop on Worst-Case Execution Time Analysis (WCET 2023)*, Vol. 114. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 1–1.

[15] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, Yutaka Ishikawa, et al. 2011. {TimeGraph}:{GPU} Scheduling for {Real-Time}{Multi-Tasking} Environments. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*.

[16] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. 2017. Replacing conjectures by positive knowledge: Inferring proven precise worst-case execution time bounds using symbolic execution. *Journal of Symbolic Computation* 80 (2017), 101–124.

[17] Manjiri K Kulkarni and JS Umale. 2016. Survey: Various Methods for WCET Estimate Calculation. *International Journal of Computer Applications* 975 (2016), 8887.

[18] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

[19] Song Liu, Jie Ma, Chenyu Zhao, Xinhe Wan, and Weiguo Wu. 2023. LFWS: Long-Operation First Warp Scheduling Algorithm to Effectively Hide the Latency for GPUs. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 106, 8 (2023), 1043–1050.

[20] Shih-Hsiang Lo, Che-Rung Lee, Quey-Liang Kao, I-Hsin Chung, and Yeh-Ching Chung. 2013. Improving GPU Memory Performancewith Artificial Barrier Synchronization. *IEEE transactions on parallel and distributed systems* 25, 9 (2013), 2342–2352.

[21] Mingsong Lv, Nan Guan, Yi Zhang, Qingxu Deng, Ge Yu, and Jianming Zhang. 2009. A survey of WCET analysis of real-time operating systems. In *2009 International Conference on embedded software and systems*. IEEE, 65–72.

[22] Xinxin Mei and Xiaowen Chu. 2016. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (2016), 72–86.

[23] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 308–317.

[24] Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. 2006. A definition and classification of timing anomalies. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)(2006)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 1–6.

[25] Moritz Sinn and Florian Zuleger. 2010. LOOPUS-A Tool for Computing Loop Bounds for C Programs.. In *WING@ ETAPS/IJCAR*. 185–186.

[26] Peter Thoman, Klaus Kofler, Heiko Studt, John Thomson, and Thomas Fahringer. 2011. Automatic OpenCL device characterization: Guiding optimized kernel design. In *European Conference on Parallel Processing*. Springer, 438–452.

[27] Robert A Van Engelen. 2001. Efficient symbolic analysis for optimizing compilers. In *International Conference on Compiler Construction*. Springer, 118–132.

[28] Xiebing Wang, Kai Huang, Alois Knoll, and Xuehai Qian. 2019. A hybrid framework for fast and accurate GPU performance estimation through source-level analysis and trace-based simulation. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 506–518.

[29] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM transactions on embedded computing systems (TECS)* 7, 3 (2008), 1–53.

[30] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 235–246.

[31] Shangshang Xiao, Mengxia Sun, Wei Zhang, Naijun Zhan, and Lei Ju. 2024. Cache Behavior Analysis with SP-Relative Addressing for WCET Estimation. In *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*. Springer, 256–274.

[32] Wei Zhang, Mingsong Lv, Wanli Chang, and Lei Ju. 2022. Precise and scalable shared cache contention analysis for WCET estimation. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 1267–1272.

[33] Shuai Zhao, Xiaotian Dai, Iain Bate, Alan Burns, and Wanli Chang. 2020. DAG scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency. In *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 128–140.